Scheme
Class outline:

- Scheme expressions
- Call expressions
- Special forms
- Examples
Scheme
A brief history of programming languages

The Lisp programming language was introduced in 1958. The Scheme dialect of Lisp was introduced in the 1970s, and is still maintained by a standards committee today.

Genealogical tree of programming languages

Scheme itself is not commonly used in production, but has influenced many other languages, and is a good example of a functional programming language.
Scheme expressions

Scheme programs consist of expressions, which can be:

- **Primitive expressions:**
  
  \[
  2 \ 3.3 \ \#t \ \#f \ + \ \text{quotient}
  \]
Scheme expressions

Scheme programs consist of expressions, which can be:

- **Primitive expressions:**
  \[ 2 \ 3.3 \ \#t \ \#f \ + \ quotient \]

- **Combinations:**
  \[(quotient \ 10 \ 2) \ (not \ #t)\]

Combinations are either a call expression or a special form.
Call expressions
Call expressions

Call expressions include an operator and 0 or more operands in parentheses:

```lisp
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
   (+ (* 2 4)
      (+ 3 5)))
  (+ (- 10 7)
     6))
```
## Built-in arithmetic procedures

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>(+ 1 2 3)</td>
</tr>
<tr>
<td>-</td>
<td>(- 12) (- 3 2 1)</td>
</tr>
<tr>
<td>*</td>
<td>(<em>) (</em> 2) (* 2 3)</td>
</tr>
<tr>
<td>/</td>
<td>(/ 2) (/ 4 2) (/ 16 2 2)</td>
</tr>
<tr>
<td>quotient</td>
<td>(quotient 7 3)</td>
</tr>
<tr>
<td>abs</td>
<td>(abs -12)</td>
</tr>
<tr>
<td>expt</td>
<td>(expt 2 10)</td>
</tr>
<tr>
<td>remainder</td>
<td>(remainder 7 3) (remainder -7 3)</td>
</tr>
</tbody>
</table>
## Built-in Boolean procedures (for numbers)

These procedures only work on numbers:

<table>
<thead>
<tr>
<th>Name</th>
<th>True expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>(= 4 4) (= 4 (+ 2 2))</td>
</tr>
<tr>
<td>&lt;</td>
<td>(&lt; 4 5)</td>
</tr>
<tr>
<td>&gt;</td>
<td>(&gt; 5 4)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>(&lt;= 4 5) (&lt;= 4 4)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>(&gt;= 5 4) (&gt;= 4 4)</td>
</tr>
<tr>
<td>even?</td>
<td>(even? 2)</td>
</tr>
<tr>
<td>odd?</td>
<td>(odd? 3)</td>
</tr>
<tr>
<td>zero?</td>
<td>(zero? 0) (zero? 0.0)</td>
</tr>
</tbody>
</table>
Built-in Boolean procedures

These procedures work on all data types:

<table>
<thead>
<tr>
<th>Name</th>
<th>True expressions</th>
<th>False expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>(eq? #t #t)</td>
<td>(eq? #t #f)</td>
</tr>
<tr>
<td></td>
<td>(eq? 0 (- 1 1))</td>
<td>(eq? 0 0.0)</td>
</tr>
<tr>
<td>not</td>
<td>(not #f)</td>
<td>(not 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(not #t)</td>
</tr>
</tbody>
</table>

The only falsey value in Scheme is #f. All other values are truthy.

Scheme procedure reference: Boolean operations

Scheme specification: Booleans
Special forms
Special forms

A combination that is not a call expression is a special form:

• if expression:
  \[(\text{if } \langle\text{predicate}\rangle \ \langle\text{consequent}\rangle \ \langle\text{alternative}\rangle)\]

• and/or:
  \[(\text{and } \langle\text{e1}\rangle \ \ldots \ \langle\text{en}\rangle)\]
  \[(\text{or } \langle\text{e1}\rangle \ \ldots \ \langle\text{en}\rangle)\]

• Binding symbols:
  \[(\text{define } \langle\text{symbol}\rangle \ \langle\text{expression}\rangle)\]

• New procedures:
  \[(\text{define } (\langle\text{symbol}\rangle \ \langle\text{formal parameters}\rangle) \ \langle\text{body}\rangle)\]

Scheme spec: special forms
define form

define <name> <expression>

Evaluates <expression> and binds the value to <name> in
the current environment. <name> must be a valid Scheme
symbol.

(define x 2)

Scheme Spec: define
define procedure

define (<name> [param] ...) <body>)

Constructs a new procedure with params as its parameters and the body expressions as its body and binds it to name in the current environment. name must be a valid Scheme symbol. Each param must be a unique valid Scheme symbol.

(define (double x) (* 2 x))

Scheme Spec: define
If expression

```
if <predicate> <consequent> <alternative>
```

Evaluates **predicate**. If true, the **consequent** is evaluated and returned. Otherwise, the **alternative**, if it exists, is evaluated and returned (if no **alternative** is present in this case, the return value is undefined).

**Example:** This code evaluates to 100/x for non-zero numbers and 0 otherwise:

```
(define x 5)
(if (zero? x)
  0
  (/ 100 x))
```
and form

(and [test] ...)

Evaluate the test s in order, returning the first false value. If no test is false, return the last test. If no arguments are provided, return #t.

Example: This and form evaluates to true whenever x is both greater than 10 and less than 20.

(define x 15)
(and (> x 10) (< x 20))

Scheme Spec: And
or form

(or [test] ...)

Evaluate the test s in order, returning the first true value. If no test is true and there are no more test s left, return #f.

Example: This or form evaluates to true when either x is less than -10 or greater than 10.

(define x -15)
(or (< x -10) (> x 10))

Scheme Spec: Or
Cond form

The cond special form that behaves similar to if expressions in Python.

```python
if x > 10:
    print('big')
elif x > 5:
    print('medium')
else:
    print('small')
```

```scheme
(cond (> x 10) (print 'big))
    ((> x 5) (print 'medium))
    (else (print 'small)))
```

```scheme
(print (cond (> x 10) 'big)
    ((> x 5) 'medium)
    (else 'small)))
```

Scheme Spec: Cond
Why is cond needed?

Without **cond**, we'd have deeply nested **if** forms:

```lisp
(if (> x 10) (print 'big)
  (if (> x 5) (print 'medium)
    (print 'small)
  ))
```

So much nicer with **cond**!

```lisp
(cond
  ((> x 10) (print 'big))
  ((> x 5) (print 'medium))
  (else (print 'small)))
```
The begin form

```python
if x > 10:
    print('big')
    print('pie')
else:
    print('small')
    print('fry')
```

```scheme
(cond (>(x 10) (begin (print 'big) (print 'pie)))
     (else (begin (print 'small) (print 'fry))))
```

Scheme Spec: Begin
The begin form

```python
if x > 10:
    print('big')
    print('pie')
else:
    print('small')
    print('fry')
```

```lisp
(cond ((> x 10) (begin (print 'big) (print 'pie)))
     (else (begin (print 'small) (print 'fry))))
```

```lisp
(if (> x 10) (begin
    (print 'big)
    (print 'pie))
    (begin
        (print 'small)
        (print 'fry)))
```

Scheme Spec: Begin
let form

The `let` special form binds symbols to values temporarily; just for one expression

```scheme
a = 3
b = 2 + 2
c = math.sqrt(a * a + b * b)
```

↑ `a` and `b` are still bound down here

```scheme
(define c (let ((a 3)
               (b (+ 2 2)))
           (sqrt (+ (* a a) (* b b)))))
```

↑ `a` and `b` are not bound down here

**Scheme Spec: Let**
**lambda expressions**

Lambda expressions evaluate to anonymous procedures.

```
(lambda ([param] ...) <body> ...)
```

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))
(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a lambda expression too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

**Scheme Spec: Lambda**
Exercises
Exercise: Sum of squares

What's the sum of the squares of even numbers less than 10, starting with some number?

Python version (iterative):

```python
def sum_of_squares(num):
    total = 0
    while num < 10:
        total += num ** 2
        num += 2
    return total

sum_of_squares(2)  # 120
Exercise: Sum of squares

What's the sum of the squares of even numbers less than 10, starting with some number?

Python version (iterative):

```python
def sum_of_squares(num):
    total = 0
    while num < 10:
        total += num ** 2
        num += 2
    return total

sum_of_squares(2)  # 120
```

Python version (recursive):

```python
def sum_of_squares(num, total):
    if num >= 10:
        return total
    else:
        return sum_of_squares(num + 2, total + num ** 2)

sum_of_squares(2, 0)  # 120
Exercise: Sum of squares (solution)

Scheme version:

(define (sum_of_squares num total)
  (if (>= num 10)
      total
      (sum_of_squares (+ num 2) (+ total (* num num)))))

(sum_of_squares 2 0)
Using helper functions

What if we said the `sum_of_squares` function could only take one argument?

In Python, we could use a helper function:

```python
def sum_of_squares(num):
    def with_total(num, total):
        if num >= 10:
            return total
        else:
            return with_total(num + 2, total + num ** 2)
    return with_total(num, 0)
```
Using helper functions (Scheme)

Similarly in Scheme!

```
(define (sum_of_squares num)
  (define (with_total num total)
    (if (>= num 10)
      total
      (with_total (+ num 2) (+ total (* num num))))
  )
  (with_total num 0)
)
```
Scheme tips

- Use the references!
  - Scheme built-in procedure
  - Scheme specification
- Auto-format your code!
- Constrain your brain: you're now living in a world of applicative programming. Look, ma, no mutation!